BIOINFORMATICS

Vol. 00 no. 00 2015
Pages 1–4

Supplemental material for MMseqs software suite for fast and deep clustering and searching of large protein sequence sets

Maria Hauser [2,*] Martin Steinegger [1,2,3,*] and Johannes Söding [1,2†]

[1]Computational Biology, Max Planck Institute for Biophysical Chemistry, Am Fassberg 11, 37077 Göttingen, Germany. [2]Gene Center, Ludwig-Maximilians-Universität München, Feodor-Lynen-Str. 25, 81377 Munich, Germany. [3]TUM, Department of Informatics, Bioinformatics & Computational Biology-I12, Boltzmannstraße 3, 85748 Garching, Germany

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Associate Editor: XXXXXXX

1 SENSITIVITY AND SPEED OF SEQUENCE SEARCHES

Our goal was to compare the performance and the speed of MMseqs with various other fast tools for protein sequence searching: SWIPE, BLAST, UBLAST, RAPsearch2 and DIAMOND.

For tools that had a limit on the maximum number of matches report, we increased this limit to ensure that at least five false positive will be listed for each query. This is necessary for the calculation of ROC5 values,

All searches were made on a computer with 128 GB RAM and two 8-core Intel Xeon E5-2680 CPUS with 2.70GHz.

MMseqs We used only the prefiltering module and the alignment module of MMseqs for the protein search. We tested two different sensitivities in the prefiltering module, $s = 4$ (default setting) and $s = 7$. Besides, we set the maximum prefiltering list length to 1000 using `--max-seqs 1000`, and the Z-score threshold to 10.0 using `--z-score-thr 10.0` in order to increase the length of the result lists for each query. The alignment module is run with the maximum e-value threshold 10.0 using `-e 10.0` and the alignment coverage is switched off using `-c 0.0`. Both modules use all 16 cores of the machine by default.

Smith-Waterman alignments with SWIPE We use the SSSE3-, multi-core-paralellized Smith-Waterman alignment calculation with SWIPE. In order to get many database matches for a query, we set the e-value to 100 using `-e 100.0` and the number of sequence descriptions and sequence alignments to 1000 using `-v 1000 -b 1000`. Additionally, swipe is instructed to use all the 16 cores of the machine with `-a 16`.

BLAST We ran BLAST using `-e 10.0` and `-v 1000 -b 1000` in order to increase the number of results, and with `-a 16` to parallelize the calculation.

UBLAST We ran UBLAST with `-evalue 10.0` option. Therefore, UBLAST outputs all significant alignments regardless of the sequence identity and alignment coverage. UBLAST uses all available cores per default. Since UBLAST does not have an option to set the maximum number of results shown, we presume that it does not have such a limit.

RAPsearch We ran RAPsearch with with `-z 16` option to paralellize the calculation, and with `-e 10.0` and `-v 1000 -b 1000` options.

DIAMOND We ran DIAMOND with `-e 10.0` option, with `--threads 16` option to paralellize the calculation, and with `--max-target-seqs 1000` option.

2 SEQUENCE CLUSTERING PERFORMANCE

We benchmarked the ability of MMseqs, blastclust, CD-HIT, kClust and USEARCH to cluster sequences based on their global similarity. We benchmarked the clustering performance with different sequence identity thresholds in the range $[0.3 : 0.7]$ in 0.1 increments. All tools were instructed to only merge sequences that had an alignment covering at least a fraction of 0.8 of the residues of both sequences.

All clustering runs (except clustering of UniProt) were made on a computer with 128 GB RAM and two 8-core Intel Xeon E5-2680 CPUS with 2.70GHz. The UniProt was clustered on a computer with 512 GB RAM and four 8-cores CPUs (Intel Xeon CPU E5-4620, 2.20GHz).

2.1 Parameters of tested tools

MMseqs We use the clustering workflow for calculating the clustering of the database. We tested simple and cascaded (option `--cascaded`) clustering each with sensitivity 4 and 7 (`-s 4` and `-s 7`) respectively. For each sensitivity, we set the target clustering sequence identity using the option `--id` and values from 0.3 to 0.7 in 0.1 increments.

blastclust Blastclust is the clustering software in the BLAST package. We set the number of used cores to 16 with the option '`-a16`', length coverage threshold to 0.8 using `-L 0.8` and the minimum sequence identity in the clusters with the option `-S` and values from 30 to 70 in 10 increments.

CD-HIT We set the minimum alignment coverage of the longer sequence to 80% with the `-aL 0.8` option and the number of threads used for the calculation to 16 with the `-T 16` option. The minimum possible clustering sequence identity in CD-HIT is 0.4. For the clustering down to the different minimum sequence identities in the range $[0.4 : 0.8]$, we used the option `-c` for the sequence identity setting and adjusted the k-mer word length with the option `-n`. For the sequence identity 0.4 the word length was set to 2, for the sequence identity 0.5 to 3, for the sequence identity 0.6 to 4 and for the sequence identity 0.7 to 5.

kClust We used the `-s` option to set the minimum sequence identity in the cluster. According to kClust recommendations, we set `-s` to 1.12, 1.73, 2.33, 2.93, 3.53, and 4.14 for the sequence identities 0.3, 0.4, 0.5, .06,

*These authors contributed equally to this work.

†to whom correspondence should be addressed: soeding@mpibpc.mpg.de

	#clusters	#seqs per cluster	#corrupted clusters
MMseqs s=4 greedy clustering	85 780	3.4	1
MMseqs s=4 set cover	60 915	4.7	1
MMseqs s=4 3-step	41 173	7.0	3
MMseqs s=7 greedy clustering	41 572	7.0	3
MMseqs s=7 set cover	29 801	9.7	2
MMseqs s=7 3-step	22 541	12.9	1
blastclust	21 890	13.3	1
CD-HIT	114 386	2.5	260
kClust	91 681	3.2	1
USEARCH	157 981	1.8	11

Table 1. Clustering results on the protein database consisting of SCOP25 and related UniProtKB sequences. Sequences put into the same cluster, but stemming from different folds are considered to be false positives.

and 0.7, respectively. Since kClust is single-threaded, we did not use any parallelization options.

USEARCH We ran USEARCH with -cluster_fast option and set the minimum sequence identity in a cluster to 50% with `--id` option to values from 0.3 to 0.8 in 0.1 increments. We set the query and target sequence coverage in USEARCH to 0.8 using the options `-query_cov 0.8` and `-target_cov 0.8`.

2.2 Cluster quality

We benchmarked the clustering quality by clustering the protein clustering benchmark dataset with the clustering tools MMseqs-sens and MMseqs-fast each with three clustering algorithms (greedy clustering as used in CD-HIT and kClust, set cover and 3-step cascaded clustering), blastclust, CD-HIT, kClust and USEARCH. We clustered the dataset down to 30% sequence identity with each tool, except for CD-HIT, where we used the minimum possible sequence identity threshold of 40%. We use the same method to define false positive sequence pairs as in the other protein search and clustering benchmarks. A cluster is considered as corrupt if it contains at least one false positive sequence pair.

All methods except CD-HIT produce clusters of very high quality with a negligible number of sequences assigned to a cluster by mistake. CD-HIT produced 260 corrupted clusters due to an occasional error in the calculation of the sequence identity. Cascaded clustering and default straight-forward clustering in MMseqs uses set-cover as the default clustering algorithm. We compared the performance of set-cover and the greedy clustering, as used by kClust. Table 1 demonstrates that set-cover performs much better.

2.3 Clustering of the UniProt database

We clustered the UniProt database version containing 54 790 250 sequences with MMseqs and USEARCH, the only two tools that are able to cluster such a large database down to sequence identities of 50% or lower. MMseqs is able to use all 32 cores for the clustering procedure, while USEARCH is able to only use one core.

We use MMseqs cascaded clustering workflow (option `--cascaded`) with default settings to evaluate the clustering procedure.

In USEARCH, we set the lowest sequence identity of clusters to 50%, since it is the lowest recommended value corresponding to the

	time	**#clusters**
blastclust	58y	?
MMseqs	8d 17h	6 374 156
USEARCH	11d 2h	9 822 910

Table 2. UniProtKB clustering results: Time and number of clusters for BLAST, MMseqs and USEARCH. kClust, BLAST and kClust times are estimated.

documentation (option `--id 0.5`). We only want to have sequences with pairwise global similarity in one cluster, so we set the query and target sequence coverage in USEARCH to 0.8 using the options `-query_cov 0.8` and `-target_cov 0.8`.

BLAST is much too slow to cluster the UniProtKB database. We estimated the runtime of BLAST clustering using a BLAST run with a small query set against the whole UniProtKB database, and extrapolated the measured runtime to the clustering of the whole UniProtKB database using an all-against-all comparison. We extrapolated that clustering based on all-against-all BLAST using all 32 cores would need about 58 years based on run times for BLAST searches of the UniProt database.

MMseqs requires 8 days and 17 hours and 118 G of memory for the clustering procedure. It produces 6 374 156 clusters, i. e. an average of $8,5$ sequences per cluster. USEARCH, on the other hand, requires, for the same job, 11 days and 2 hours and 42 GB of memory, while it produces 9 822 910 clusters, i. e. an average of $5,5$ sequences per cluster. The results are shown in Table 2.

Although MMseqs is parallelized and uses all 32 cores of the computer and USEARCH is single-threaded, USEARCH runs almost as fast. This is in part explained by the efficiency of the greedy agglomerative clustering algorithm used by USEARCH, which reduces the total number of comparisons from N_{seqs}^2 to $N_{\text{clus}}N_{\text{seqs}}$, where N_{seqs} is the number of sequences in the database and N_{clues} is the number of representative sequences, i.e., the number of clusters in the clustered database. However, loosing a factor $N_{\text{seqs}}/N_{\text{clues}}$ in speed over the simple greedy algorithm is more than counterbalanced by the possibility to parallelize the set-cover clustering and by its superior clustering performance in comparison to the simple greedy algorithm.

3 PREFILTERING ALGORITHM DETAILS

3.1 Prefilter Z-score

Expected prefiltering scores between non-homologous sequences will be proportional to the product of both their lengths, since there is a small but non-negligible probability for any pair of query-target k-mers to attain a similarity score above the k-mer cut-off score. It makes sense to correct for the score expected from such background k-mer matches by subtracting it from the actual score. The background score may also depend on the amino acid distribution of the query sequence and on whether it contains regions with strongly biased amino acid composition, as these regions can cause many k-mer matches with unrelated sequences containing similarly biased regions.

Instead of simply estimating the background score from the product of the lengths of query and target sequences, we measure the expected k-mer score of the query sequence *per column of a target sequence*. For that purpose, we can assume that the overwhelming majority of the database sequences are not homologous to a given query sequence.

For each query, perform a calibration search through a database of 100 000 randomly sampled target sequences, whose sum of lengths

$$\text{sum}_L = \sum_{t=1}^{N}(L_t - k + 1)$$

we recored. We then sum up all prefiltering scores for query sequence q with the database,

$$\text{sum}_S \quad = \quad \sum_{t=1}^{N} S_{qt} \,,$$

where N is the database size, L_t is the length of the database sequence t and S_{qt} is the prefiltering score of the query sequence q with a database sequence t. Then, the expected chance prefiltering score between q and a target database sequence t is

$$S_0 \quad = \quad (L_t - k + 1) \, \frac{\text{sum}_\text{S}}{\text{sum}_\text{L}}$$

We also correct for the lower score relative dispersion at high lengths by dividing S_0 by the estimate of its standard deviation. We assume that the number of k-mer matches is Poisson-distributed, and the standard deviation of the score should therefore be proportional to the square root of the number of expected k-mer matches, which is

$$n_\text{match} \quad \approx \quad (L_t - k + 1) \, \frac{\text{sum}_\text{S}}{\text{sum}_\text{L}} / S_\text{match} \,,$$

where S_match is the expected score per chance k-mer match. Under the assumption that the number of k-mer matches is Poisson-distributed, the standard deviation of the chance score of the query sequence with a database sequence t is

$$\begin{aligned} \sigma_S \quad &= \quad \sqrt{n_\text{match}} \, S_\text{match} \\ &= \quad \sqrt{(L_t - k + 1) \, \frac{\text{sum}_\text{S}}{\text{sum}_\text{L}} \, S_\text{match}} \end{aligned}$$

Therefore, the offset- and scale-corrected score Z_{qt} for a query sequence q and a database sequence t is

$$Z_{qt} \quad = \quad \frac{S_{qt} - S_0}{\sigma_S} \,.$$

We are interested in all sequence pairs, where $Z_{qt} > Z_{thr}$, i. e. the prefiltering score should fulfill the condition

$$\begin{aligned} S_{qt} \quad &\geq \quad Z_{thr} \, \sigma_S + S_0 \\ &\geq \quad Z_{thr} \sqrt{(L_t - k + 1) \, \frac{\text{sum}_\text{S}}{\text{sum}_\text{L}} S_\text{match}} + (L_t - k + 1) \, \frac{\text{sum}_\text{S}}{\text{sum}_\text{L}} \,. \end{aligned}$$

Calculating the offset- and scale-corrected score threshold for each pair q, t would slow down the retrieving of prefiltering results. Since the threshold value $Z_{thr} \, \sigma_S + S_0$ depends only on the length of the database sequence t for a fixed q, the database sequences are ordered by length, the threshold is calculated once and recalculated only if the length of the next sequence falls below 95% of the reference sequence length.

The statistical analysis of the scores gets unreliable for small databases of fewer than 100 000 sequences, since in this case there will not be enough sequences to calculate the expected score and the standard deviation reliably. We use pseudo-counts to make the estimate of S_0 and σ_S robust. We define the size of the pseudo-database as 100 000 with an average sequence length 350 (average sequence length in UniProtKB) and an average sequence composition. We estimate k-mer match probability and set the score of a chance k-mer match S_match to be slightly above the k-mer similarity threshold. k-mer match probability estimation is explained in detail in section Automatic sensitivity setting. Then, we calculate n_{match}, sum_S and sum_L by adding the pseudo counts to the empirical counts.

3.2 Amino acid local composition bias correction

Some sequences have regions of low complexity with an amino acid composition that differs considerably from the background amino acid distribution assumed in the amino acid substitution matrix. Low complexity regions of a sequence can lead to high prefiltering scores sequences containing similarly biased low complexity regions. To alleviate this effect, we correct for the local compositional bias in the sequences by assigning lower scores to the matches of *locally* frequent amino acids. We examine $d = 20$ amino acids on both sides of the the amino acid x_i at position i in the sequence. Score correction ΔS_i at position i is

$$\Delta S_i(x_i) \quad = \quad -\frac{1}{2d} \sum_{j=i-d, j \neq i}^{i+d} S(x_i, x_j) + \sum_{a=1}^{20} f(a) S(a, x_i)$$

where $S(x_i, x_j)$ is the amino acid substitution score between amino acids x_i and x_j, and $f(a)$ is the background frequency of the amino acid a. Then, the final corrected score S_c for the match of x_i with another amino acid y_j is

$$S_c(x_i, y_j) \quad = \quad S(x_i, y_j) + \Delta S_i(x_i)$$

Therefore, amino acids that are less frequent in the window $\pm d$ around the sequence position i than the background frequency of this amino acid contribute more to the score, and the more frequent less.

4 PARALLELIZATION WITH OPENMP

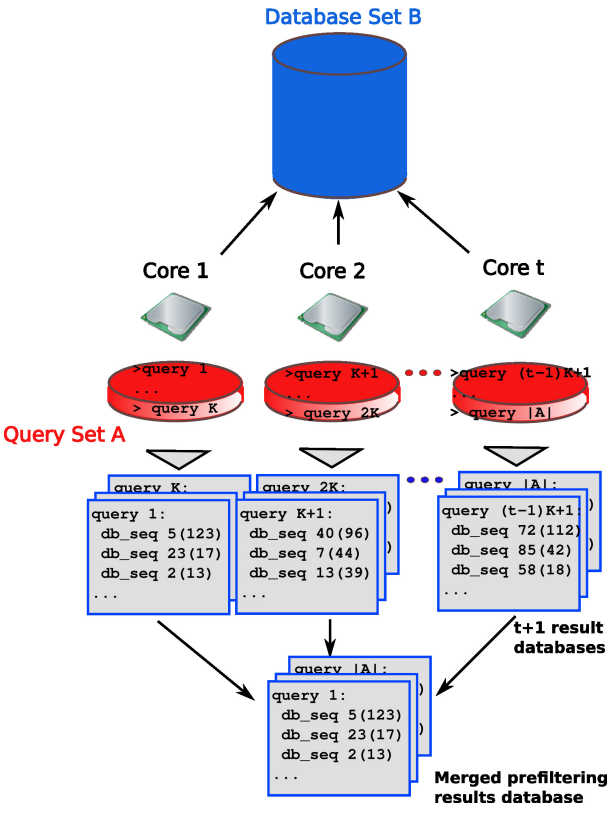

Fig. 1. Parallelization scheme of the prefilter module using OpenMP.

The parallelization approach of the prefiltering and alignment modules is shown in Fig. 1 . Parts of the query sequence set are matched against the database in parallel. Each thread writes to its own output database. In the end, all results are merged into one output database.

We benchmarked the multicore scaling performance of the MMseqs prefilter module with the dataset used in the speed measurements. We tested the run time behavior with five different threads settings 1, 2, 4, 8 and 16

shown in Fig. 2. The prefilter performance scales nearly linearly up to 4 threads. Beyond this the performance scales sublinearly because of the high amount of random memory accesses to the loops 3 and 4 of Figure 1.

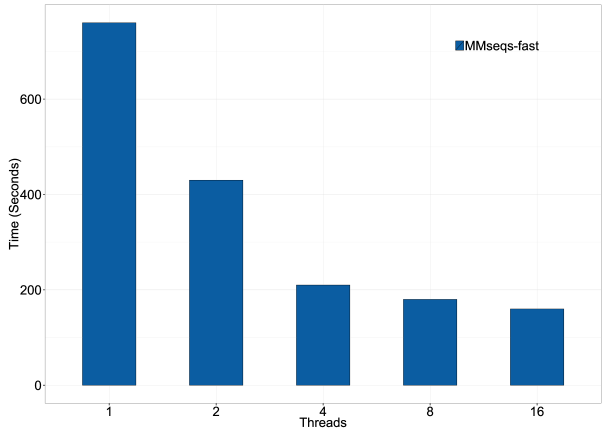

Fig. 2. Multithread scaling of the MMseqs prefilter.. Runtime in seconds for the MMseqs prefilter to run 7616 query sequences against 54 790 250 sequences on different threads setting.

5 CLUSTERING RESULTS

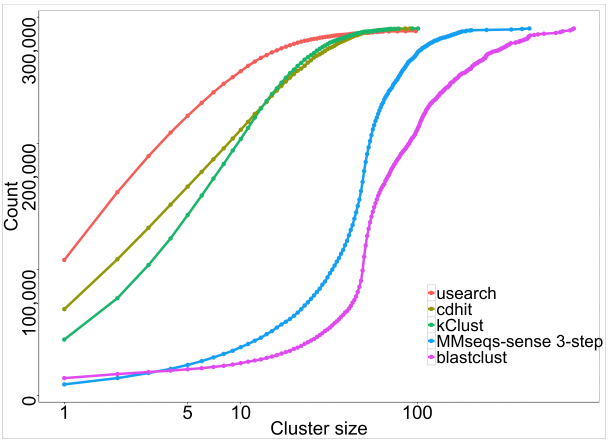

Fig. 3. Cumulative cluster size distributions of blastclust, MMseqs, kClust, CD-HIT, usearch for a clustering threshold of 0.3, corresponding to the leftmost point in Fig. 3F.

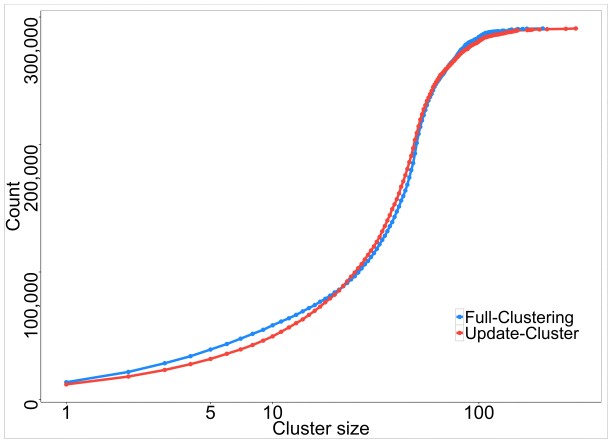

Fig. 4. Cumulative cluster size distribution after ten updating steps versus after a single cascaded clustering, using the parameter -s 7.